FOUR APPLICATIONS OF A SOFTWARE DATA COLLECTION AND ANALYSIS METHODOLOGY

Victor R. Basili [1] and Richard W. Selby, Jr. [2]

[1] Department of Computer Science, University of Maryland, College Park, MD 20742, USA

[2] Department of Information and Computer Science, University of California, Irvine, CA 92717, USA; was with the Department of Computer Science, University of Maryland, College Park, MD 20742, USA

ABSTRACT

The evaluation of software technologies suffers because of the lack of quantitative assessment of their effect on software development and modification. A seven-step data collection and analysis methodology couples software technology evaluation with software measurement. Four in-depth applications of the methodology are presented. The four studies represent each of the general categories of analyses on the software product and development process: 1) blocked subject-project studies, 2) replicated project studies, 3) multi-project variation studies, and 4) single project studies. The four applications are in the areas of, respectively, 1) software testing strategies, 2) Cleanroom software development, 3) characteristic software metric sets, and 4) software error analysis.

1. Introduction

Software management decisions and research need to be based on sound analysis and criteria. However, it seems that many decisions and issues are resolved by inexact means and seasoned judgment, without the support of appropriate data and analysis. Problem formulation coupled with the collection and analysis of appropriate data is pivotal to any management, control, or quality improvement process, and this awareness motivates our investigation of the analysis processes used in software research and management. Our objectives for this work, which updates [14], include 1) structuring the process of analyzing software technologies, 2) investigating particular goals and questions in software development and modification, 3) characterizing the use of quantitative methods in analysis of software, and 4) identifying problem areas of data collection and analysis in software research and management.

Section 2 outlines a seven-step methodology for data collection and analysis. Section 3 discusses coupling the formulation of goals and questions with quantitative analysis methods. The application of the data collection and analysis paradigm in four empirical studies is presented in Section 4. Section 5 identifies several problem areas of data collection and analysis in software research and management. Section 6 presents a summary of this paper.

2. Methodology for Data Collection and Analysis

Several techniques and ideas have been proposed to improve the software development process and the delivered product. There is little hard evidence, however, of which methods actually contribute to quality in software development and modification. As the software field emerges, the need for understanding the important factors in software production continues to grow. The evaluation of software technologies suffers because of the lack of quantitative assessment of their effect on software development and modification.

This work supports the philosophy of coupling methodology with measurement. That is, tying the processes of software methodology use and evaluation together with software measurement. The assessment of factors that affect software development and modification is then grounded in appropriate measurement, data analysis, and result interpretation. This section describes a quantitatively based approach to evaluating software technologies. The formulation of problem statements in terms of goal/question hierarchies is linked with measurable attributes and quantitative analysis methods. These frameworks of goals and questions are intended to outline the potential effect a software technology has on aspects of cost and quality.

The analysis methodology described provides a framework for data collection, analysis, and quantitative evaluation of software technologies. The paradigm identifies the aspects of a well-run analysis and is intended to be applied in different types of problem analysis from a variety of problem domains. The methodology presented serves not only as a problem formulation and analysis paradigm, but also suggests a scheme to characterize analyses of software development and modification. The use of the paradigm highlights several problem areas of data collection and analysis in software research and management.

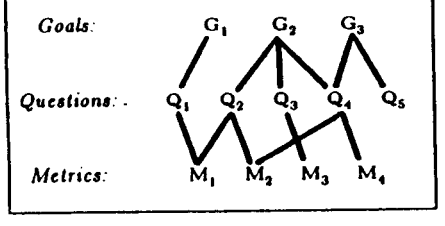
Presented at the NATO Advanced Study Institute: The Challenge of Advanced Computing Technology to System Design Methods Durham, United Kingdom, July 29-August 10, 1985. (Will appear in a book by Springer-Verlag.)

The methodology described for data collection and analysis has been applied in a variety of problem domains and has been quite useful. The methodology consists of seven steps that are listed below and discussed in detail in the following paragraphs (see also [14, 16]). 1) Formulate the goals of the data collection and analysis. 2) Develop a list of specific questions of interest. 3) Establish appropriate metrics and data categories. 4) Plan the layout of the investigation, experimental design, and statistical analysis. 5) Design and test the data collection forms or automated collection scheme. 6) Perform the investigation concurrently with data collection and validation. 7) Analyze and interpret the data in terms of the goal/question framework.

A first step in a management or research process is to define a set of goals. Each goal is then refined into a set of sub-goals that will contribute to reaching that goal. This refinement process continues until specific research questions and hypotheses have been formulated. Associated with each question are the data categories and particular metrics that will be needed in order to answer that question. The integration of these first three steps in a goal/question/metric hierarchy (see Figure 1) expresses the purpose of an analysis, defines the data that needs to be collected, and provides a context in which to interpret the data.

In order to address these research questions, investigators undertake several types of analyses. Through these analyses, they attempt to substantially increase their knowledge and understanding of the various aspects of the questions. The analysis process is then the basis for resolving the research questions and for pursuing the various goals. Before actually collecting the data, the data analysis techniques to be used are planned. The appropriate analysis methods may require an alternate layout of the investigation or additional pieces of data to be collected. A well planned investigation facilitates the interpretation of the data and generally increases the usefulness of the results.

Figure 1. Goal/question/metric paradigm.



Once it is determined which data should be gathered, the investigators design and test the collection method. They determine the information that can be automatically monitored, and customize data collection forms to the particular environment. After all the planning has occurred, the data collection is performed concurrently with the investigation and is accompanied by suitable data validity checks.

As soon as the data have been validated, the investigators do preliminary data analysis and screening using scatter plots and histograms. After fulfilling the proper assumptions, they apply the appropriate statistical and analytical methods. The statistical results are then organized and interpreted with respect to the goal/question framework. More information is gathered as the analysis process continues, with the goals being updated and the whole cycle progressing.

3. Coupling Goals With Analysis Methods

Several of the steps in the above data collection and analysis methodology interrelate with one another. The structure of the goals and questions should be coupled with the methods proposed to analyze the data. The particular questions should be formulated to be easily supported by analysis techniques. In addition, questions should consider attributes that are measurable. Most analyses make some result statement (or set of statements) with a given precision about the effect of a factor over a certain domain of objects. Considering the form of analysis result statements will assist the formation of goals and questions for an investigation, and will make the statistical results more readily correspond to the goals and questions.

3.1. Forms of Result Statements

Consider a question in an investigation phrased as "For objects in the domain **D**, does factor **F** have effect **S**?". The corresponding result statement could be "Analysis **A** showed that for objects in the domain **D**, factor **F** had effect **S** with certainty **P**.". In particular, a question could read "For novice programmers doing unit testing, does functional testing uncover more faults than does structural testing?". An appropriate response from an analysis may then be "In a blocked subject-project study of novice programmers doing unit testing, functional testing

domain from which they were obtained. Thus as the size of the sampled domain and the degree to which it represents other populations increase, the wider-reaching the conclusion.

The next section cites several software analyses from the literature and classifies them according to this scheme pictured in Figure 2. Segments of four examinations in different analysis categories will then be presented.

3.3. Analysis Classification and Related Work

Several investigators have published studies in the four general areas of blocked subject-project [17, 28, 29, 33, 36, 37, 38, 49, 55, 64, 70, 71], replicated project [4, 10, 21, 25, 34, 43, 44, 45, 46, 47, 52, 56, 60, 62, 63], multi-project variation [1, 3, 8, 11, 12, 18, 20, 22, 24, 66, 67, 68], and single project [2, 5, 9, 13, 15, 19, 32, 35, 54, 57]. Study overviews appear in [23, 51, 59, 61].

4. Application of the Methodology

The following sections briefly describe four different types of studies in which the data collection and analysis methodology described above has been applied. The particular analyses are 1) a blocked subject-project study comparing software testing strategies, 2) a replicated project study characterizing the effect of using the Cleanroom software development approach, 3) a multi-project variation study determining a characteristic set of software cost and quality metrics, and 4) a single project study examining the errors that occurred in a medium-size software development project.

4.1. Software Testing Strategy Comparison

After first giving an overview of the study, this section describes the software testing techniques examined, the investigation goal/question framework, the experimental design, analysis, and major conclusions.

4.1.1. Overview and Major Results

To demonstrate that a particular program actually meets its specifications, professional software developers currently utilize several different testing methods. An empirical study comparing three of the more popular techniques (functional testing, structural testing, and code reading) has been conducted with 32 professional programmers as subjects. In a fractional factorial design, the individuals applied each of the three testing methods to three different programs containing faults. The formal statistical approach enables the distinction among differences in the testing techniques, while allowing for the effects of the different experience levels and programs. The major results from this study of junior, intermediate, and advanced programmers doing unit testing are the following. 1) Code readers detected more faults than did those using the other techniques, while functional testers detected more faults than did structural testers. 2) Code readers had a higher fault detection rate than did those using the other methods, while there was no difference between functional testers and structural testers. 3) The number of faults observed, fault detection rate, and total effort in detection depended on the type of software tested. 4) Subjects of intermediate and junior expertise were not different in number of faults found or fault detection rate, while subjects of advanced expertise found a greater number of faults than did the others, but were not different from the others in fault detection rate. 5) Code readers and functional testers both detected more omission faults and more control faults than did structural testers, while code readers detected more interface faults than did those using the other methods.

4.1.2. Testing Techniques

Figure 3 shows the different capabilities of the three software testing techniques of code reading, functional testing, and structural testing. In functional testing, which is a "black box" approach [41], a programmer constructs test data from the program's specification through methods such as equivalence partitioning and boundary value analysis [53]. The programmer then executes the program and contrasts

Figure 3. Capabilities of the testing methods.

	code reading	functional testing	structural testing
view program specification	X	X	X
view source code	X		X
execute program		X	X

uncovered more faults than did structural testing ($\alpha < .05$).".

Result statements on the effects of factors have varying strengths, but usually are either characteristic, evaluative, predictive, or directive. Characteristic statements are the weakest. They describe how the objects in the domain have changed as a result of the factor. E.g., "A blocked subject-project study of novice programmers doing unit testing showed that using code reading detected and removed more logic faults than computation faults ($\alpha < .05$)." Evaluative statements associate the changes in the objects with a value, usually on some scale of goodness or improvement. E.g., "A blocked subject-project study of novice programmers doing unit testing showed that using code reading detected and removed more of the expensive faults to correct than did functional testing ($\alpha < .05$)." Predictive statements are a stronger statement type. They describe how objects in the domain *will* change if subjected to a factor. E.g., "A blocked subject-project study showed that for novice programmers doing unit testing, the use of code reading *will* detect and remove more logic faults than computation faults ($\alpha < .05$)." Directive statements are the strongest type. They foretell the value of the effect of applying a factor to objects in the domain. E.g., "A blocked subject-project study showed that for novice programmers doing unit testing, the use of code reading *will* detect and remove more of the expensive faults to correct than will functional testing ($\alpha < .05$)." The analysis process then consists of an investigative procedure to achieve the result statements of the desired strength and precision after considering the nature of the factors and domains involved.

Given any factor, researchers would like to make as strong a statement of as high a precision about the factor's effect in as large a domain as possible. Unfortunately, as the statement applies to an increasingly large domain, the strength of the statement or the precision with which we can make it may decrease. In order for analyses to produce useful statements about factors in large domains, the particular aspects of a factor and the domains of its application must be well understood and incorporated into the investigative scheme.

3.2. Analysis Categorization

Two important sub-domains that should be considered in the analysis of factors in software development and modification are the individuals applying the technology and what they are applying it to. These two sub-domains will loosely be referred to as the "subjects," a collection of (possibly multi-person) teams engaged in separate development efforts, and the "projects," a collection of separate problems or pieces of software to which a technology is applied. By examining the sizes of these two sub-domains ("scopes of evaluation") considered in an analysis, we obtain a general classification of analyses of software in the literature.

Figure 2 presents the four part analysis categorization scheme. Blocked subject-project studies examine the effect of possibly several technologies as they are applied by a set of subjects on a set of projects. If appropriately configured, this type of study enables comparison within the groups of technologies, subjects, and projects. In replicated project studies, a set of subjects may separately apply a technology (or maybe a set of technologies) to the same project or problem. Analyses of this type allow for comparison within the groups of subjects and technologies (if more than one used). A multi-project variation study examines the effect of one technology (or maybe a set of technologies) as applied by the same subject across several projects. These analyses support the comparison within groups of projects and technologies (if more than one used). A single project analysis involves the examination of one subject applying a technology on a single project. The analysis must partition the aspects within the particular project, technology, or subject for comparison purposes.

Result statements of all four types mentioned above can be derived from all these analysis classes. However, the statements will need to be qualified by the

Figure 2. Categorization of Analyses of Software

#Teams per project	#Projects	
	one	more than one
one	Single project	Multi-project variation
more than one	Replicated project	Blocked subject-project

its actual behavior with that indicated in the specification. In structural testing, which is a "white box" approach [40, 42], a programmer inspects the source code and then devises test cases based on the percentage of the program's statements executed (the "test set coverage") [65]. The structural tester then executes the program on the test cases and compares the program's behavior with its specification. In code reading by stepwise abstraction [48, 50], a person identifies prime subprograms in the software, determines their functions, and then composes these functions to determine a function for the entire program. The code reader then compares this derived function and the specifications (the intended function).

4.1.3. Investigation Goals

The goals of this study comprise four different aspects of software testing: fault detection effectiveness, fault detection cost, classes of faults detected, and effect of programmer expertise level. A framework of the goals and specific questions appears in Figure 4.

Figure 4. Structure of goals/subgoals/questions for testing experiment. (Each of these questions should be prefaced by "For junior, intermediate, and advanced programmers doing unit testing, ... ".)

I. Fault detection effectiveness
  A. Which of the testing techniques (code reading, functional testing, or structural testing) detects the greatest number of faults in the programs?
    1. Which of the techniques detects the greatest percentage of faults in the programs (the programs each contain a different number of faults)?
    2. Which of the techniques exposes the greatest number (or percentage) of program faults (faults that are observable but not necessarily reported)?
  B. Is the number (or percentage) of faults observed dependent on the type of software?

II. Fault detection cost
  A. Which of the testing techniques has the highest fault detection rate (number of faults detected per hour)?
  B. Which of the testing techniques requires the least amount of fault detection time?
  C. Is the fault detection rate dependent on the type of software?

III. Classes of faults detected
  A. Do the methods tend to capture different classes of faults?
  B. What classes of faults are observable but go unreported?

IV. Effect of programmer expertise level
  A. Does the performance of junior, intermediate, and advance programmers differ in any of the above goal categories?

4.1.4. Experimental Design

Admittedly, the goals stated here are quite ambitious. It is not implied that this experiment can definitively answer all of these questions. The intention, however, is to gain insights into their answers and into the merit and appropriateness of each of the techniques.

A fractional factorial experimental design was employed in the analysis [27]. There were three testing techniques, three programs containing faults, and three levels of programmer expertise. Each subject used each technique and tested each program, while not testing a given program more than once. The analysis of variance model included the two-way and three-way interactions among the main effects, and nested the random effect of subjects within programmer expertise.

The programs were representative of three different classes of software: a text formatter (also appeared in [52]), an abstract data type, and a database maintainer [38]. The programs had 169, 147, and 365 lines of high-level source code, respectively, and were a realistic size for unit testing. They had nine, seven, and twelve faults, respectively, which were intended to be representative of commonly occurring software faults [69].

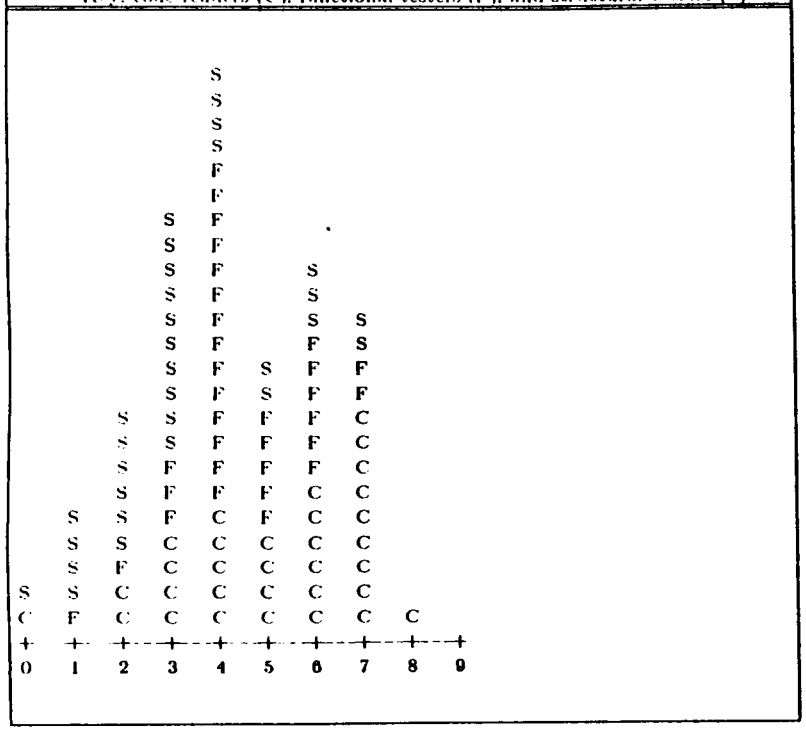
The subjects were professional programmers from NASA Goddard and Computer Sciences Corporation, a major NASA contractor. They had an average of 10 years professional experience (SD = 5.7).

For a complete description of the programs, faults, subjects, experimental operation, and analysis see [17, 59].

4.1.5. Data Analysis

Segments of the data analysis and interpretation for two of the goal areas appear in the following sections. Figure 5 displays the number of faults in the programs detected by the techniques.

Figure 5.
Number of faults detected in the programs.
Key: code readers (C), functional testers (F), and structural testers (S).



4.1.5.1. Fault Detection Effectiveness

The subjects applying code reading detected an average of 5.09 (SD = 1.92) faults per program, persons using functional testing found 4.47 (SD = 1.34), and those applying structural testing uncovered 3.25 (SD = 1.80); the subjects detected an overall average of 4.27 (SD = 1.86) faults per program. The overall F-test that the techniques detected the same number of faults (Question I.A) was rejected (α = .0001; the probability of Type I error is reported). Subjects using code reading detected 1.24 more faults per program than did subjects using either functional or structural testing ($\alpha < .0001$, 95% confidence interval 0.73 - 1.75). Subjects using functional testing detected 1.11 more faults per program than did those using structural testing ($\alpha < .0007$, 95% c.i. 0.52 - 1.70). Since the programs each had a different number of faults, an alternate interpretation compares the percentage of the programs' faults detected by the techniques (Question I.A.1). The techniques performed in the same order when percentages are compared: subjects applying code reading detected 10.0% more faults per program than did subjects using the other techniques ($\alpha < .0001$, c.i. 9.0 - 22.1%), and subjects applying functional testing detected 11.2% more faults than did those using structural testing ($\alpha < .003$, c.i. 4.1 - 18.3%). Thus comparing either the number or percentage of faults detected, individuals using code reading observed the most faults, persons applying functional testing found the second most, and those doing structural testing uncovered the fewest.

4.1.5.2. Fault Detection Cost

The subjects applying code reading detected faults at an average rate of 3.33 (SD = 3.42) faults per hour, persons using functional testing found faults at 1.84 (SD = 1.00) faults per hour, and those applying structural testing uncovered faults at a rate of 1.82 (SD = 1.21) faults per hour; the subjects detected faults at an overall average rate of 2.33 (SD = 2.28) faults per hour. The overall F-test that the techniques detected faults at the same rate (Question II.A) was rejected ($\alpha < .0014$). Subjects using code reading detected 1.49 more faults per hour than did subjects using either functional or structural testing ($\alpha < .0003$, c.i. 0.75 - 2.23). Subjects using functional and structural testing were not statistically different in fault detection rate ($\alpha > .05$). The subjects spent an average of 2.75 (SD = 1.57) hours per program detecting faults. Comparing the total time spent in fault detection, the techniques were not statistically different ($\alpha > .05$). Thus, subjects using code reading detected faults at a higher rate than did those applying functional or structural testing, while the total fault detection effort was not different among the methods.

4.1.6. Research Summary

The strategies of functional testing, structural testing, and code reading are compared in four different aspects of software testing: fault detection effectiveness, fault detection cost, classes of faults detected, and effect of programmer expertise level. The major results of this study appear in the earlier "Overview and Major Results" section.

Presented in this section are some fundamental features and results of an empirical study. The results given are from a sample of junior, intermediate, and advanced programmers applying the techniques to given programs with particular faults. The direct extrapolation of these findings to other testing environments is not implied. However, valuable insights into improving the effectiveness of software testing have been gained. For a complete presentation of the study, see [17, 59].

4.2. Cleanroom Development Approach Analysis

After first giving an overview of the study, this section describes the Cleanroom software development approach, the investigation goals, a replicated project study applying the Cleanroom approach, the analysis of its effect relative to a more traditional approach, and the conclusions.

4.2.1. Overview and Major Results

The Cleanroom software development approach is intended to produce highly reliable software by integrating formal methods for specification and design, complete off-line development, and statistically-based testing. In an empirical study, fifteen teams developed versions of the same software system (800 - 2300 source lines); ten teams applied Cleanroom, while five applied a more traditional approach. This analysis characterizes the effect of Cleanroom on the delivered product, the software development process, and the developers. The major results from this study of teams of novice and intermediate programmers building a small system are 1) most developers were able to effectively apply the techniques of Cleanroom; 2) the Cleanroom teams' products more completely met system requirements and had a higher percentage of successful test cases; 3) the source code developed using Cleanroom had more comments and less dense complexity; 4) the use of Cleanroom successfully modified development style; and 5) most Cleanroom developers indicated they would use the approach again.

4.2.2. Cleanroom Software Development

The need for highly reliable software and for discipline in the software development process motivates the Cleanroom software development approach. In addition to improving the control during development, this approach is intended to deliver a product that meets several quality aspects: a system that conforms with the requirements, a system with high operational reliability, and source code that is easily readable and modifiable.

The Federal Systems Division of IBM [30, 31] presents the Cleanroom software development method as a technical and organizational approach to developing software with certifiable reliability. The idea is to deny the entry of defects during the development of software, hence the term "Cleanroom." The focus of the method is imposing discipline on the development process by integrating formal methods for specification and design, complete off-line development, and statistically-based testing. With the intention that correctness is "designed" into the software rather then "tested" in, developers are not allowed to test their own programs. They focus on off-line review techniques, such as code reading, inspections, and walkthroughs, to assert the correctness of their system. Independent testers then simulate the operational environment of the product with functional testing, record observed failures, and determine an objective measure of system reliability.

4.2.3. Investigation Goals

Some intriguing aspects of the Cleanroom approach include 1) development without testing and debugging of programs, 2) independent program testing for quality assurance (rather than to find faults or to prove "correctness" [39]), and 3) certification of system reliability before product delivery. In order to understand the effects of Cleanroom, the goals of this investigation are to I) characterize the effect of Cleanroom on the delivered product, II) characterize the effect of Cleanroom on the software development process, and III) characterize the effect of Cleanroom on the developers. An example question under goal I would be "For teams of novice and intermediate programmers building a small system, does Cleanroom deliver a product more completely meeting its requirements than does a traditional development approach?". A complete framework of goals and questions for this study appears in [60].

4.2.4. Empirical Study Using Cleanroom

In order to pursue the above goals, an empirical study was executed comparing team projects developed using Cleanroom with those using a more conventional approach. Fifteen three-person teams each developed an approximately 1200 line electronic mail system over a six week period at the University of Maryland. The subjects had an average of 1.7 years professional experience. Ten three-person teams

applied Cleanroom, while five applied a more traditional approach. The other aspects of the development were the same.

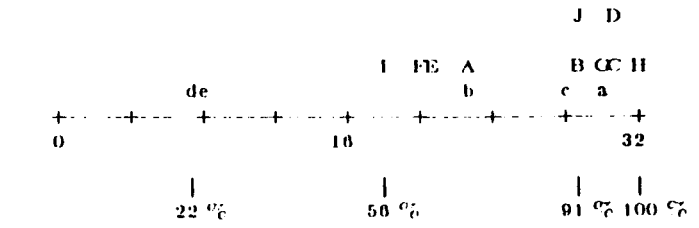
4.2.5. Data Analysis

Segments of the analysis and interpretation of the data collected in the study appear in the following sections, organized by the goal areas outlined earlier. The systems developed by the various teams ranged from 824 to 2284 source lines, from 410 to 999 executable statements, and from 18 to 67 procedures.

4.2.5.1. Characterization of the Effect on the Product Developed

Completeness of implementation was examined as one contrast among the operational properties of the systems delivered by the two groups (Question I.A.1). A measure of implementation completeness was calculated by partitioning the required system into sixteen logical functions (e.g., send mail to an individual, read a piece of mail, respond, add yourself to a mailing list, ...). Each function in an implementation was then assigned a value of two if it completely met its requirements, a value of one if it partially met them, or zero if it was inoperable. The total for each system was calculated; a maximum score of 32 was possible. Figure 6 displays this subjective measure of requirement conformance for the systems (Cleanroom teams in upper case; the significance levels for the Mann-Whitney statistics reported are the probability of Type I error in an one-tailed test). A first observation is that six of the ten Cleanroom teams built very close to the entire system. While not all of the Cleanroom teams performed equally well, a majority of them applied the approach effectively enough to develop nearly the whole product. More importantly, the Cleanroom teams met the requirements of the system more completely than did the non-Cleanroom teams.

Figure 6. Requirement conformance of the systems.



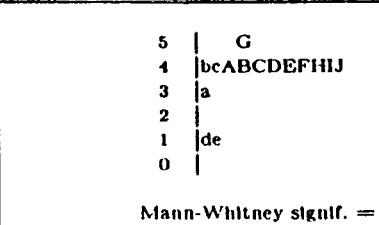
Mann-Whitney signif. = .088

In summary of the effect on the product, Cleanroom developers delivered a product that 1) more completely met system requirements, 2) had a higher percentage of successful operationally-based test cases, and 3) had more comments and less dense complexity.

4.2.5.2. Characterization of the Effect on the Development Process

Schedule slippage continues to be a problem in software development. It would be interesting to see whether the Cleanroom teams demonstrated any more discipline by maintaining their original schedules (Question II.C). All of the teams from both groups planned four releases of their evolving system, except for team 'G' which planned five. At each delivery an independent party would operationally test the functions currently available in the system, according to the team's implementation plan. In Figure 7, we observe that all the teams using Cleanroom kept to their original schedules by making all scheduled deliveries.

Figure 7.
Number of system releases.



Mann-Whitney signif. = .006

Summarizing the effect on the development process, Cleanroom developers 1) felt they more effectively applied off-line review techniques, while non-Cleanroom teams focused on functional testing; 2) spent less time on-line and used fewer computer resources; and 3) tended to make all their scheduled deliveries.

4.2.5.3. Characterization of the Effect on the Developers

The first question posed in this goal area is whether the individuals using Cleanroom missed the satisfaction of executing their own programs (Question III.A). Figure 8 presents the responses to a question included in the postdevelopment attitude survey on this issue. As might be expected, almost all the individuals missed some aspect of program execution. As might not be expected, however, this missing of program execution had no relation to several product quality measures.

Figure 8.
Breakdown of responses to the attitude survey question, "Did you miss the satisfaction of executing your own programs?".

13 Yes, I missed the satisfaction of program execution.
11 - I somewhat missed the satisfaction of program execution.
4 - No, I did not miss the satisfaction of program execution.

In summary of the effect on the developers, most Cleanroom developers 1) modified their development style, 2) missed program execution, and 3) indicated they would use the approach again.

4.2.6. Research Summary

This section describes a study of "Cleanroom" software development – an approach intended to produce highly reliable software by integrating formal methods for specification and design, complete off-line development, and statistically-based testing. This is the first investigation known to the authors that applied Cleanroom and characterized its effect relative to a more traditional development approach. The major results of this study appear in the earlier "Overview and Major Results" section.

This empirical study is intended to advance the understanding of the relationship between introducing discipline into the development process (as in Cleanroom) and several aspects of product quality (conformance with requirements, high operational reliability, and easily modifiable source code). A more complete description of the application of the Cleanroom in the study, the data analysis, and the conclusions appears in [50, 60].

4.3. Characteristic Software Metric Set Study

After first giving an overview of the study, this section briefly describes a characteristic software metric set, the investigation goals, empirical study, and data analysis.

4.3.1. Overview

In software development and maintenance, several metrics have been proposed to predict product cost/quality and to capture distinct project aspects. Since both cost/quality goals and production environments differ, this study examines an approach for customizing a characteristic set of software metrics to an environment. The approach is applied in the Software Engineering Laboratory (SEL), a NASA Goddard production environment [6, 7, 20, 58]. The uses examined for a characteristic metric set include forecasting the effort for development, modification, and fault correction of modules based on historical data.

4.3.2. Characteristic Software Metric Sets

A characteristic software metric set is a concise collection of measures that capture distinct factors in a software development/maintenance environment. A characteristic metric set could be used to 1) characterize an environment, 2) compare an environment with others, 3) monitor current project status, or 4) forecast project outcome relative to past projects, when metrics in the set are available early in development.

4.3.3. Investigation Goals

The goals for this investigation are to I) develop an approach for customizing a set of measures to particular cost/quality goals in a particular environment, II) calculate the metric set for the NASA/S.E.L. environment, and III) examine the usability of the approach as a management tool. An example question under Goal III would be "In the NASA/S.E.L. environment of projects and programmers, does determining a characteristic metric set and using historical data enable one to predict which

modules will be difficult to change?". A goal/question framework appears in [18, 59].

4.3.4. Empirical Study

A proposed approach for calculating a characteristic set consists of three steps: 1) formulate the goals and questions that represent cost/quality factors in an environment, 2) list all measures that capture information relating to the goals, and 3) condense the measures into a set capturing distinct factors (e.g., by using factor analysis). This approach has been applied to six projects from the NASA/S.E.L. environment consisting of 652 newly developed modules. The projects range from 51,000 to 112,000 lines of FORTRAN and from 6000 to 22,300 person-hours of development.

4.3.5. Data Analysis

The particular goals chosen for the NASA/S.E.L. environment are to analyze system development effort, system faults, and system modifications. A total of 49 candidate process and product metrics were examined. The use of principal factor analysis isolated the characteristic metric set for the NASA/S.E.L. environment: {source lines, fault correction effort per executable statement, design effort, code effort, number of I/O parameters, number of versions}.

4.3.6. Research Summary

This study investigates an approach for customizing a set of software metrics to an environment. A characteristic software metric set is intended to help support the effective management of software development and maintenance. The approach examined for building a characteristic metric set will be adaptable to different cost/quality goals and different environments. A more complete presentation of this study appears in [18].

4.4. Software Error Analysis

This section first gives an overview of the software error study, then describes the error classification schemes, investigation goals, software project examined, data analysis, and conclusions.

4.4.1. Overview and Major Results

Insights into the characterization and improvement of software development can be gained through analysis of the types of errors made during software projects. This study analyzes various relationships involving the errors occurring in one project from the NASA/SEL production environment. The major results from this study of a team of intermediate and advanced programmers building a medium-scale system are the following. 1) A majority of the errors were due to incorrect or misinterpreted functional specifications or requirements. 2) The larger a module was, the less error-prone it tended to be. 3) Errors contained in modified modules required more effort to correct than did those in new modules. 4) Although both new and modified modules had a high percentage of interface errors, new modules had a higher percentage of control errors, while modified modules had a higher percentage of data and initialization errors. 5) The error data reflects that the developers were involved in a new application with changing requirements.

4.4.2. Error Classification

Several classification and distribution schemes were applied to the errors observed during the project. Two abstract classification schemes characterize the types of software errors. One error categorization method separates errors of omission from errors of commission. A second error categorization scheme partitions software errors into the six classes of 1) initialization, 2) computation, 3) control, 4) interface, 5) data, and 6) cosmetic. A third approach distinguishes among errors based on the source of the error during development, e.g., incorrect or misinterpreted requirements, design error involving several modules, misunderstanding of the external environment, etc. An explanation of these classification schemes appears in [13]. These classification schemes are intended to distinguish among different reasons that programmers make errors in software development.

4.4.3. Investigation Goals

There are three goal areas in this investigation. I.) Characterize the frequency and distribution of errors that occurred during development. An example question would be "What are the sources of the errors that occur?". II.) Analyze the relationships between the frequency and distribution of errors and various environmental factors. An example question here would be "How does the reuse of existing design

and code relate to the effort required for error correction?". III.) Interpret the results of the analysis relative to other software error studies. "Are the distributions of error types in this project similar to errors in projects from the same (SEL) environment?" would be an example question under this goal area. Each of the above questions should be prefaced by "When a team of intermediate and advanced programmers builds a medium-size software project, ... ".

4.4.4. Empirical Study

The software project analyzed in the study was developed in the Software Engineering Laboratory (SEL), a NASA Goddard production environment. The system developed was a general-purpose program for satellite planning studies and was approximately 90,000 lines of FORTRAN. The error data was collected over a period of 33 months, including the development phases of coding, testing, acceptance, and maintenance. The system represents a new application for the developers, even though it uses several algorithms similar to those in other SEL projects. Consequently, the system requirements kept growing and changing more than would be expected in a typical ground-support software project.

4.4.5. Data Analysis

Segments of the analysis and interpretation of the data collected in the study appear in the following sections, organized by the goal areas outlined earlier.

4.4.5.1. Characterize the Frequency and Distribution of Errors

In the system of 370 modules, there were a total of 215 errors found during the development, which were contained in 90 (26%) of the modules. Of the modules found to contain errors, 51% were newly developed modules and 49% were modified from previous projects. Figure 9 presents a distribution of the errors based on their source. Sixty five percent of the errors was attributed to incorrect or misinterpreted functional specifications or requirements. Two thirds of the functional specification errors were in modules modified from systems with different applications. Therefore, although these modules had the desired basic function, it appears that they were not well-enough specified to be reused under slightly different circumstances.

Figure 9. Distribution of Project Errors by Source.

Error Source	%Total
Requirements incorrect or misinterpreted	16
Functional specification incorrect or misinterpreted	49
Design error involving several modules	5
Error in design or implementation of a single module	30
Misunderstanding of external environment	0
Error in the use of the programming language or compiler	0

4.4.5.2. Analyze the Relationships Between Errors and Environmental Factors

Although modifying modules from previous systems may reduce the amount of coding effort, developers need to consider the effort required to correct any errors in the modified modules. Figure 10 displays the distribution of effort required for error correction in both new and modified modules. Forty five percent of the errors required at least one day to correct. Of the errors needing at least one day to correct, a higher percentage were in modified modules (27% of total) as opposed to newly developed modules (18% of total). In general, errors occurring in newly developed modules needed less effort to correct than did those in modified modules.

4.4.5.3. Interpretation of the Errors Relative to Other Projects

Figure 11 compares the distribution of error sources in this project with another project developed in the SEL environment [89]. The other project shown is a typical SEL ground-support software project with a representative distribution of faults.

Figure 10. Distribution of Error Correction Effort for New and Modified Modules.

Correction Effort	%New	%Modified	%Total
1 hour or less	21	15	36
1 hour to 1 day	11	8	19
1 day to 3 days	3	15	18
more than 3 days	15	12	27

The overall distributions of errors sources are not very similar. In the typical SEL project, the project requirements tend to be stable and the appropriate high-level designs are pretty well understood. Hence, the majority of errors are localized within single modules. In the current project examined, the frequent change of the system requirements is reflected in a higher proportion of errors associated with incorrect or misinterpreted requirements.

4.4.6. Research Summary

This section describes an analysis of the errors found during a medium-scale software project. The study is intended to increase the understanding of relationships among types of errors, environmental factors, and project characteristics. A more complete presentation of the error analysis in this software project appears in [13].

5. Problem Areas

From the use of the data collection and analysis methodology, we identify several problem areas in data collection and analysis in software research and management. 1) The process of formulating intuitive problems into precisely stated goals is a nontrivial task. The inherent difficulty in goal writing reflects the uncertainty of all aspects of quality in the software product and development process. 2)

Figure 11. Comparison of Error Source Distributions.

Error Source	Current Project (%)	Another SEL Project (%)
Requirements incorrect or misinterpreted	16	5
Functional specification incorrect or misinterpreted	49	3
Design error involving several modules	5	10
Error in design or implementation of a single module	30	72
Misunderstanding of external environment	0	1
Error in the use of the programming language or compiler	0	8
Other	0	1

Numerous software metrics have been proposed to measure distinct attributes of software. These metrics need to be validated to determine whether they actually capture what is intended. 3) The process of collecting accurate data is a continuing challenge. While there is increasing potential in automated collection schemes, the more common data collection forms are subject to incompleteness, inconsistency, and human error. 4) There have been a growing number of controlled experiments done to determine which factors contribute to software quality. In order for the results of these studies to apply to other environments, the samples (of programmers, programs, ...) must be of sufficient size and be representative of production environments. 5) These controlled studies are expensive to conduct. Both industry and academia must help support these efforts; e.g., academic researchers using subjects from industry. 6) There seems to be an interdependency among several factors that contribute to product and process quality. The use of several techniques together may be effective as a "critical mass," making the isolation of their individual effects difficult. 7) The methods of analysis must account for the high variation in individual performance. Without careful planning, the many-to-one differential among humans can taint experimental results. 8) Researchers have rarely been able to reproduce results across environments. In addition to the lack of consistent use of measures, every software development or modification environment seems to differ.

6. Summary

Problem formulation coupled with the collection and analysis of appropriate data is pivotal to any quality improvement process. This work investigates various problem analysis approaches that are relevant to software research and management. A seven-step data collection and analysis methodology was described that has been feasible and useful in a variety of problem domains. Aspects of the approach include the use of the goal/question/metric paradigm, and the need to couple proposed goals and questions with measurable attributes and appropriate analysis methods. We presented the goal structure, analysis, and preliminary conclusions for segments of four different types of studies in which the analysis methodology is applied: a blocked subject-project study comparing software testing strategies, a replicated project study characterizing the effect of using the Cleanroom software development approach, a multi-project variation study determining a characteristic set of software cost and quality metrics, and a single project study examining the errors that oc-

curred in a medium-size software development project. In addition to exhibiting a research methodology and a spectrum of software analyses, these empirical studies are intended to advance the understanding of 1) the contribution of various software testing strategies to the software development process and to one another; 2) the relationship between introducing discipline into the development process and several aspects of product quality (conformance with requirements, high operational reliability, and easily modifiable source code); 3) the use of software metrics to characterize environments and predict project cost/quality outcome; and 4) the relationships between software error types and various aspects of development. Finally, we identified several problem areas in data collection and analysis in software research and management.

7. Acknowledgement

Research supported in part by the Air Force Office of Scientific Research Contract AFOSR-F49620-80-C-001 and the National Aeronautics and Space Administration Grant NSG-5123 to the University of Maryland. Computer support provided in part by the Computer Science Center at the University of Maryland.

8. References

[1] E. N. Adams, Optimizing Preventive Service of Software Products, *IBM Journal of Research and Development* **28**, 1, pp. 2-14, Jan. 84.

[2] J.-L. Albin and R. Ferreol, Collecte et analyse de mesures de logiciel (Collection and Analysis of Software Data), *Technique et Science Informatiques* **1**, 4, pp. 297-313, 1982. (Rairo ISSN 0752-4072)

[3] J. W. Bailey and V. R. Basili, A Meta-Model for Software Development Resource Expenditures, *Proc. Fifth Int. Conf. Software Engr.*, San Diego, CA, pp. 107-116, 1981.

[4] J. W. Bailey, Teaching Ada: A Comparison of Two Approaches, Dept. Com. Sci., Univ. Maryland, College Park, MD, working paper, 1984.

[5] F. T. Baker, System Quality Through Structured Programming, *AFIPS Proc. 1972 Fall Joint Computer Conf.* **41**, pp. 339-343, 1972.

[6] V. R. Basili, M. V. Zelkowitz, F. E. McGarry, R. W. Reiter, Jr., W. F. Truszkowski, and D. L. Weiss, The Software Engineering Laboratory, Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD, Rep. SEL-77-001, May 1977.

[7] V. R. Basili and M. V. Zelkowitz, Analyzing Medium-Scale Software Developments, *Proc. Third Int. Conf. Software Engr.*, Atlanta, GA, pp. 116-123, May 1978.

[8] V. R. Basili and K. Freburger, Programming Measurement and Estimation in the Software Engineering Laboratory, *Journal of Systems and Software* **2**, pp. 47-57, 1981.

[9] V. R. Basili and D. M. Weiss, Evaluation of a Software Requirements Document By Analysis of Change Data, *Proc. Fifth Int. Conf. Software Engr.*, San Diego, CA, pp. 314-323, March 9-12, 1981.

[10] V. R. Basili and R. W. Reiter, A Controlled Experiment Quantitatively Comparing Software Development Approaches, *IEEE Trans. Software Engr* **SE-7**, May 1981.

[11] V. R. Basili and C. Doerflinger, Monitoring Software Development Through Dynamic Variables, *Proc. COMPSAC*, Chicago, IL, 1983.

[12] V. R. Basili, R. W. Selby, Jr., and T. Y. Phillips, Metric Analysis and Data Validation Across FORTRAN Projects, *IEEE Trans. Software Engr.* **SE-9**, 6, pp. 652-663, Nov. 1983.

[13] V. R. Basili and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation, *Communications of the ACM* **27**, 1, pp. 42-52, Jan. 1984.

[14] V. R. Basili and R. W. Selby, Jr., Data Collection and Analysis in Software Research and Management, *Proceedings of the American Statistical Association and Biometric Society Joint Statistical Meetings*, Philadelphia, PA, August 13-16, 1984.

[15] V. R. Basili and J. R. Ramsey, Structural Coverage of Functional Testing, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1442, Sept. 1984.

[16] V. R. Basili and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data*, *Trans. Software Engr.* **SE-10**, 6, pp. 728-738, Nov. 1984.

[17] V. R. Basili and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep., 1985. (submitted to the *IEEE Trans. Software Engr.*)

[18] V. R. Basili and R. W. Selby, Jr., Calculation and Use of an Environment's Characteristic Software Metric Set, *Proc. Eighth Int. Conf. Software Engr.*, London, August 28-30, 1985.

[19] V. R. Basili, E. E. Katz, N. M. Panlilio-Yap, C. L. Ramsey, and S. Chang, A Quantitative Characterization and Evaluation of a Software Development in Ada, *IEEE Computer*, September 1985.

[20] B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[21] B. W. Boehm, T. E. Gray, and T. Seewaldt, Prototyping Versus Specifying: A Multiproject Experiment, *IEEE Trans. Software Engr.* **SE-10**, 3, pp. 290-303, May 1984.

[22] J. Bowen, Estimation of Residual Faults and Testing Effectiveness, *Seventh Minnowbrook Workshop on Software Performance Evaluation*, Blue Mountain Lake, NY, July 24-27, 1984.

[23] R. E. Brooks, Studying Programmer Behavior: The Problem of Proper Methodology, *Communications of the ACM* **23**, 4, pp. 207-213, 1980.

[24] W. D. Brooks, Software Technology Payoff: Some Statistical Evidence, *J. Systems and Software* **2**, pp. 3-9, 1981.

[25] F. O. Buck, Indicators of Quality Inspections, IBM Systems Products Division, Kingston, NY, Tech. Rep. 21.802, Sept. 1981.

[26] D. N. Card, F. E. McGarry, J. Page, S. Eslinger, and V. R. Basili, The Software Engineering Laboratory, Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD Rep. SEL-81-104, Feb. 1982.

[27] W. G. Cochran and G. M. Cox, *Experimental Designs*, John Wiley & Sons, New York, 1950.

[28] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics, *IEEE Trans. Software Engr.*, pp. 96-104, March 1979.

[29] B. Curtis, S. B. Sheppard, and P. M. Milliman, Third Time Charm: Stronger Replication of the Ability of Software Complexity Metrics to Predict Programmer Performance, *Proc. Fourth Int. Conf. Software Engr.*, pp. 356-360, Sept. 1979.

[30] M. Dyer and H. D. Mills, Developing Electronic Systems with Certifiable Reliability, *Proc. NATO Conf.*, Summer, 1982.

[31] M. Dyer, Cleanroom Software Development Method, IBM Federal Systems Division, Bethesda, MD, October 14, 1982.

[32] A. Endres, An Analysis of Errors and their Causes in Systems Programs, *IEEE Trans. Software Engr.*, pp. 140-149, June 1975.

[33] J. D. Gannon and J. J. Horning, The Impact of Language Design on the Production of Reliable Software, *Trans. Software Engr.* **SE-1**, pp. 179-191, 1975.

[34] J. D. Gannon, An Experimental Evaluation of Data Type Conventions, *Communications of the ACM* **20**, 8, pp. 584-595, 1977.

[35] J. D. Gannon, E. E. Katz, and V. R. Basili, Characterizing Ada Programs: Packages, *The Measurement of Computer Software Performance*, Los Alamos National Laboratory, Aug. 1983.

[36] J. D. Gould and P. Drongowski, An Exploratory Study of Computer Program Debugging, *Human Factors* **16**, 3, pp. 258-277, 1974.

[37] J. D. Gould, Some Psychological Evidence on How People Debug Computer Programs, *International Journal of Man-Machine Studies* **7**, pp. 151-182, 1975.

[38] W. C. Hetzel, An Expermental Analysis of Program Verification Methods, Ph.D. Thesis, Univ. of North Carolina, Chapel Hill, 1976.

[39] W. E. Howden, Reliability of the Path Analysis Testing Strategy, *IEEE Trans. Software Engr.* **SE-2**, 3, Sept. 1976.

[40] W. E. Howden, Algebraic Program Testing, *Acta Informatica* **10**, 1978.

[41] W. E. Howden, Functional Program Testing, *IEEE Trans. Software Engr.* **SE-6**, pp. 162-169, Mar. 1980.

[42] W. E. Howden, A Survey of Dynamic Analysis Methods, pp. 209-231 in *Tutorial: Software Testing & Validation Techniques, 2nd Ed.*, ed. E. Miller and W. E. Howden, 1981.

[43] D. H. Hutchens and V. R. Basili, System Structure Analysis: Clustering With Data Bindings, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1310, August 1983.

[44] S-S. V. Hwang, An Empirical Study in Functional Testing, Structural Testing, and Code Reading/Inspection, Dept. Com. Sci., Univ. of Maryland, College Park, Scholarly Paper 362, Dec. 1981.

[45] W. L. Johnson, S. Draper, and E. Soloway, An Effective Bug Classification Scheme Must Take the Programmer Into Account, *Proc. Workshop High-Level Debugging*, Palo Alto, CA, 1983.